CS388G Problem Set #2

Nidhi Kadkol (nk9368)

February 2019

1

The algorithm is as follows: Arrange the edges in increasing order of weights, and use Kruskal's algorithm to find a minimal spanning tree. Let the weight of this tree be w_{min}. If $w_{min} > k$, then G does not have a spanning tree of weight k, and we return false from the algorithm. If $w_{min} = k$, then we return true. Now since $w_{min} < k$, we arrange the edges in decreasing order of weights, and use Kruskal's algorithm of adding the edges in this order to find a maximal spanning tree. Let the weight of this graph be w_{max}. If $k \leq w_{max}$, then we return true, else we return false.

To show the correctness of this algorithm, we show that if $w_{min} \leq k \leq w_{max}$, then we can construct a spanning tree of weight k.

Proof. If $k = w_{min}$ or $k = w_{max}$, then the spanning tree we can construct is the minimal or the maximal respectively. If $w_{min} < k < w_{max}$, then we first take the minimal spanning tree T of weight w_{min}. We take the list of the edges of weight 1 that do not belong to T. Note that the number of such weight 1 edges is $\geq w_{max} - w_{min}$. We take the first edge from the list and add it to T. Since all the nodes in T are connected, adding this edge will create a cycle. To remove this cycle, we remove an edge of weight 0 from T. This increases the weight of T by 1, and it still remains a spanning tree, as by removing an edge from the cycle, we are still ensuring that all the nodes remain connected. If there is no edge of weight 0 in the cycle, then we don't add that edge and we proceed to the next edge in the list and try to add it in the same manner. We guarantee that we will reach an edge of weight 1 which can be added to the graph such that it creates a cycle which has at least one edge of weight 0 that we will remove to break the cycle. If this is not possible, then it means that we have gone through the entire list of edges with weight 1 and we are unable to add any of them to T. This means that we cannot increase the weight of T by adding any edge of weight 1 and removing an edge of weight 0. This would mean that T is the maximal spanning tree, that is, we cannot increase its weight further. So the weight of T is equal to w_{max}. But we know that the weight of T is w_{min}, which is a contradiction. So we will definitely find an edge to add to T to get a new spanning tree of weight $w_{min} + 1$. Now, we repeat the same procedure to continue adding edges to the tree until we reach the spanning tree of weight k. To show this is possible, we use the same vein of reasoning as above. Suppose we have a spanning tree T of weight $j < w_{max}$. Then there exists an edge of weight 1 that is not in T, and when we add this edge to T we will get a cycle which has at least one edge of weight 0.

By removing this 0 weight edge, we get a tree of weight $j+1$. If we couldn't find any edge of weight 1 that can be added in this way, it means that none of the weight 1 edges can be added, and thus we cannot increase the weight of T. Thus T is a maximal spanning tree. But this means that the weight of T must be w_{max}, which is a contradiction. Also, we will definitely have at least $w_{max} - j$ edges to add, since we know that a spanning tree of weight w_{max} exists. ■

The time it takes to sort the edges and carry out Kruskal's algorithm is $O(E\log(E))$, since we do this twice our runtime is $O(E\log(E))$ which is polynomial time complexity.

2

2(a)

A schedule for unit-time tasks with deadlines and penalties for n tasks has a task scheduled in every time slot from 1 to n. This algorithm makes a greedy choice of choosing the task with the highest penalty at any point and placing it as detailed in the algorithm. We show that making this choice at every step will always lead to an optimal answer i.e. a schedule with the lowest possible total penalty.

First, we argue that if we place the first activity using this greedy choice, then there exists an optimal solution that has made this choice.

Proof. Consider a set S sorted into monotonically decreasing order by penalty. Let a_1 be the first task in this set, that is a_1 has the highest penalty w_1 among all the tasks in S. According to the algorithm, we place a_1 in the latest available time slot before its deadline. Since no time slots have been filled yet and for all i, $1 \leq d_i \leq n$, such a time slot exists for a_1. Let us call this time slot t_1. Now we show that an optimal solution exists with a_1 in slot t_1.

Let us suppose there is an optimal schedule with total penalty P where a_1 is not in slot t_1. Then there is another task a_i with penalty $w_i < w_1$ in slot t_1.

Case 1: a_1 is in a slot after t_1.

This means that a_1 is late, as t_1 is the latest slot before d_1. If a_i is also late, then we can swap a_i and a_1. a_1 will now be early, and a_i will still be late. So the new penalty $P' = P - w_1 < P$. If a_i is not late, then making the swap could either result in a_i being late or a_i remaining early. If a_i remains early, then $P' = P - w_1 < P$. If a_i becomes late, then $P' = P - w_1 + w_i < P$, as $w_1 > w_i$. Thus, we will get a schedule with lower penalty contradicting the assumption that the schedule without a_1 in t_1 was optimal.

Case 2: a_1 is in a slot before t_1.

Here, a_1 is early and the schedule incurs no penalty from a_1. If a_i is also early, then swapping a_1 and a_i will make no difference to P as a_i is being moved to an earlier time slot so it is still early, and a_1 is being moved to t_1 which is before d_1. So if the schedule with P is optimal, the schedule after swapping is also optimal. If a_i is late, then it swapping with a_1 could either make a_i early or a_i could still be late. If it is still late, then there is no change in the penalty, and if it becomes early then the new penalty will be $P' = P - w_i < P$, leading us to a better solution than the original one, and so the initial schedule was not optimal.

Thus, if there exists an optimal schedule, then there always exists an optimal schedule which has the greedy choice made first. ∎

Now we show that making the greedy choice for the remaining tasks at every step leads us to an optimal solution.

Proof. Consider a set S sorted into monotonically decreasing order by penalty. Suppose we are considering task a_i in this ordering. There are 2 cases for deciding where to place a_i:

Case 1: There exists an empty slot k just before d_i.

In this case, we place a_i in slot k. Now we justify that this choice leads to an optimal solution. Suppose this choice does not give us an optimal solution. Then in an optimal solution with penalty P there is a different task a_j in slot k with $w_j < w_i$ and a_i is either in a slot after k or a slot before k. We examine these 2 cases separately.

Case 1.1: a_i is placed in a slot after k. This means that a_i is late. If a_j is also late, then we can swap a_i and a_j. a_j will still be late but now a_i will be early. The new penalty $P' = P - w_i < P$. so this contradicts the claim that the original schedule was optimal. If a_j is early, then swapping a_i and a_j will lead to a_j becoming late or a_j remaining early. If a_j remains early, then the new penalty $P' = P - w_i < P$. If a_j becomes late, then it incurs a penalty $w_j < w_i$. So the new penalty $P' = P - w_i + w_j < P$. Thus, by swapping we will always get a lower penalty then the original one, so the claim that given an optimal schedule exists, placing a_i before k leads to an optimal schedule is false.

Case 1.2: a_i is placed in a slot before k.

This means that a_i is early. If a_j is early as well, then swapping a_i and a_j will not change the total penalty, as a_j is moved to an earlier slot and a_i is moved to slot k which is before its deadline. Thus, if the original solution was optimal, this is an optimal solution as well. If a_j is late, then swapping a_i and a_j will either make a_j early or not change the fact that it is late, while a_i will continue to be early. If swapping makes no change in the lateness of a_j, then the total penalty does not change and this is an optimal solution as well, since the original solution is assumed to be optimal. If a_j becomes early, then the new penalty of the schedule is $P' = P - w_j < P$. Thus, we get a better solution when we swap a_i and a_j and our claim that the original solution was optimal is false.

Thus, if there is an empty slot k before d_i, then there exists an optimal solution where a_i is placed in slot k.

Case 2: There is no empty slot before d_i.

In this case, we will place a_i in the latest unfilled slot l. Now we show that there exists an optimal solution where this choice is made.

Consider an optimal solution with total penalty P where a_i is not placed in l. Since l is the latest available slot, it means that a_i is placed in some slot r earlier than l, and a different task a_j with penalty $w_j < w_i$ is placed in l. Also, a_i is late, as there are no available slots before d_i. Suppose we swap a_i and a_j. Let this new schedule have penalty P'. Since a_i is moved to a later slot, it will continue to be late, and not have an effect on the penalty of the new schedule. There are 3 cases for a_j in the original schedule before the swap.

Case 2.1: a_j is early.
In this case, swapping moves a_j to an earlier slot, so it continues to be early, and $P' = P$. Since the original schedule was optimal, this schedule is also optimal.

Case 2.2: a_j is late, and $d_j < r$.
In this case, a_j is still late after swapping, so $P' = P$. Since the original schedule was optimal, this schedule is also optimal.

Case 2.3: a_j is late, and $d_j > r$. In this case, swapping a_i and a_j will make a_j change from being late to being early, so the new penalty $P' = P - d_j < P$. So the original schedule cannot be optimal.

This shows that any any given step, if we are considering an activity according to the monotonically decreasing order of penalties, then making the greedy choice always leads to an equal or better solution than making any other choice. Thus, this is an optimal choice at any step of the algorithm, and making this choice at every step leads us to an optimal solution. ∎

2(b)

Placing a task a_i in its slot takes constant time. Computing which slot to place it in would naively take $O(n)$ time to (find the latest available slot before its deadline or to find the last unfilled spot), but we can reduce this using the fast disjoint-set forest implementation. We use a set to represent a set of tasks assigned at contiguous times. The representative of the set will have additional *high* and *low* attributes which denote the earliest and latest time of that set of tasks. We implement the functions as follows:

Algorithm 1 Make-Set function for disjoint-set forest

```
1: function MAKE-SET($a$)
2:     $a.p = a$
3:     $a.rank = 0$
4:     return $a$
5: end function
```

Algorithm 2 Find-Set function for disjoint-set forest

```
function FIND-SET($x$)
    if $x \neq x.p$ then
        $x.p = $ FIND-SET($x.p$)
    end if
    return $x.p$
end function
```

Algorithm 3 Union function for disjoint-set forest

```
function UNION($a, b$)
    $x = $ FIND-SET($a$)
    $y = $ FIND-SET($b$)
    $l = \min(x.low, y.low)$
    $h = \max(x.high, y.high)$
    if $x.rank > y.rank$ then
        $y.p = x$
        $x.low = l$
        $x.high = h$
    else
        $x.p = y$
        $y.low = l$
        $y.high = h$
        if $x.rank == y.rank$ then
            $y.rank = y.rank + 1$
        end if
    end if
end function
```

We initialize an empty array D of size n, where $D[i]$ points to the activity that is scheduled in time-slot i. We use the above functions in our algorithm as illustrated below:

Algorithm 4 Scheduling unit-time tasks using fast disjoint-set forest

```
for $i \leftarrow 1$ to $n$ do
    if $D[a_i.deadline] == NULL$ then        ▷ If spot just before deadline is free
        $spot = a_i.deadline$
    else
        $b = $ FIND-SET$(D[a_i.deadline])$
        if $b.low == 0$ then
            $spot = b.low - 1$        ▷ Find latest unfilled spot before deadline
        else
            if $D[n] == NULL$ then        ▷ If last spot is free
                $spot = n$
            else
                $c = $ FIND-SET$(D[n])$
                $spot = c.low - 1$        ▷ Find latest unfilled spot
            end if
        end if
    end if
    $x = $ MAKE-SET$(a_i)$
    $x.low = spot$
    $x.high = spot$
    $D[spot] = x$
    if $spot \neq 0$ then
        if $D[spot - 1] \neq NULL$ then
            UNION$(D[spot - 1], D[spot])$
        end if
    end if
    if $spot \neq n$ then
        if $D[spot + 1] \neq NULL$ then
            UNION$(D[spot], D[spot + 1])$
        end if
    end if
end for
```

Our loop runs n times, and in each iteration the function FIND-SET is called at most twice, MAKE-SET is called once, and UNION is called once. MAKE-SET has constant run time, and UNION and FIND-SET have a very slow worst case running time of $\alpha(n)$. Thus, our implementation's running time is $O(n\alpha(n))$.

3

Let X and Y belong to $\mathcal{I}$ with $|X| < |Y|$. Then there exists a matching A that matches X and a matching B that matches Y. A and B are both sets of disjoint edges. Consider the subgraph $A \oplus B$. It can either be a null set or a set of disjoint simple paths and even length cycles. If it is a null set, then X and Y have the same matching $A = B$. Since $|Y| > |X|$, there is a vertex $y \in Y \setminus X$ that is matched by A. So we can add y to X, and this set will have the matching A.

If $A \oplus B$ is not a null set then since $|X| < |Y|$ there is at least one odd-length path that begins and ends with an edge that matches a vertex in Y. The endpoints y_1 and y_2 of this path are not matched by A. We see that the matching $A \oplus P$ matches all the vertices in $X + y_1$ and $X + y_2$, so we can add either y_1 or y_2 to X and this set will have the matching $A \oplus P$. Thus, the exchange property holds.

4

$$\begin{aligned}
&M = (S, \mathcal{I}) \\
&X = \text{basis of } M = \{x_1, x_2, \ldots, x_n\} \\
&x \in S \setminus X \\
&Y = X + x \\
&C = \text{set of all } y \text{ in } Y \text{ such that } (Y - y) \in \mathcal{I}
\end{aligned}$$

4(a)

First, note that

$$\begin{aligned}
&X \in \mathcal{I} \\
&(X + x) - x \in \mathcal{I} \\
&Y - x \in \mathcal{I} \\
\implies &x \in C
\end{aligned}$$

Let $C = X' + x$ where $X' \subseteq X$. We have separate cases for X', and in each of these, we shall show that $C \notin \mathcal{I}$.

Case 1: $X' = X$
In this case,

$$\begin{aligned}
C &= X' + x \\
&= X + x
\end{aligned}$$

Since X is a basis, it is a maximal independent subset of S. So if we add any element to X, the resulting set will not be independent. That is,

$$\begin{aligned}
&X + x \notin \mathcal{I} \\
\implies &\ C \notin \mathcal{I}
\end{aligned}$$

Case 2: X' is some proper subset of X.

Let $|X'| = m$, where m is a non-negative integer. Then

$$\begin{aligned}|C| &= |X' + x| \\ &= m + 1\end{aligned}$$

We also know that $|X| = n$. Since X' is a proper subset of C, $m < n$, or $m + 1 <= n$.

Case 2.1: $m + 1 = n$

We prove by contradiction that $C \notin \mathcal{I}$. Suppose, if possible, $C \in \mathcal{I}$. Since $m = n - 1$, there is one element $x_k \in X$ that does not belong to X'. All the other $m = n - 1$ elements of X are in X'. Thus,

$$\begin{aligned}X' &= X - x_k \\ C &= X' + x \\ &= X - x_k + x \\ &= (X + x) - x_k \\ &= Y - x_k\end{aligned}$$

Since $C \in \mathcal{I}$, $Y - x_k \in \mathcal{I}$. But by definition, this means that $x_k \in C$. This is a contradiction, as we know that X' does not contain x_k and hence C does not contain x_k. Thus, our assumption that $C \in \mathcal{I}$ is incorrect. Hence, $C \notin \mathcal{I}$.

Case 2.2: $m + 1 < n$

We prove by contradiction that $C \notin \mathcal{I}$. Suppose, if possible, $C \in \mathcal{I}$. Since $|C| < |X|$, then by exchange property we can add elements to C from $X \setminus C$ until we get a new set C' such that $|C'| = |X|$, and $C' \in \mathcal{I}$. The number of elements we add is $|X| - |C| = n - m - 1$. Since $C = X' + x$ with $X' \subset X$ and $x \notin X$, thus $X \setminus C = X \setminus X'$. The number of elements in $X \setminus X'$ is $|X| - |X'| = n - m$. Since we are adding $(n - m) - 1$ elements from $X \setminus X'$ to C, we are adding all except one element from $X \setminus X'$. Let this element be x_i. Then,

$$\begin{aligned}C' &= C + X \setminus X' - x_i \\ &= X' + x + X - X' - x_i \\ &= X + x - x_i \\ &= Y - x_i\end{aligned}$$

Since $C' \in \mathcal{I}$, $Y - x_i \in \mathcal{I}$. But by definition, this means that $x_i \in C$. This is a contradiction, as we know that X' does not contain x_i and hence C does not contain x_i. Thus, our assumption that $C \in \mathcal{I}$ is incorrect. Hence, $C \notin \mathcal{I}$.

4(b)

C is equal to x + some (possibly empty) subset of X. Let us denote this subset of X by X'. So,

$$\begin{aligned} C &= X' + x, \quad \text{where } X' \subseteq X \\ C - y &= X' + x - y, \quad \text{for all } y \text{ in } C \\ &\subseteq X + x - y \\ &\subseteq Y - y \end{aligned}$$

Since $Y - y \in \mathcal{I}$, therefore by hereditary property $C - y \in \mathcal{I}$.

4(c)

Let $B = \{b_1, b_2, \ldots, b_k\}$ be a subset of Y.

Then $Z = Y - B$, where $Z \notin \mathcal{I}$. We now claim that there is no element of B that belongs to C, ie. there is no b_i such that $b_i \in C$.

Proof. We prove this by contradiction. Suppose there exists a b_i such that b_i is an element of C.

By definition, $Y - b_i \in \mathcal{I}$. Also, since $B \subseteq Y$,

$$\begin{aligned} &(B - b_i) \subseteq (Y - b_i) \\ \Longrightarrow\ &(Y - b_i) - (B - b_i) \subseteq (Y - b_i) \\ \Longrightarrow\ &(Y - B) \subseteq (Y - b_i) \\ \Longrightarrow\ &(Y - B) \in \mathcal{I} \ \text{ (by hereditary property)} \\ \Longrightarrow\ &Z \in \mathcal{I} \end{aligned}$$

Which is a contradiction. Hence, there is no element of B that belongs to C. ■

Since C is a set of all y in Y such that $Y - y \in \mathcal{I}$, thus $C \subseteq Y$.
Since no element of C belongs to B, thus

$$\begin{aligned} &C \subseteq Y - B \\ \Longrightarrow\ &C \subseteq Z \end{aligned}$$

Hence, C is contained in Z.

5

17-5(d)

- Suppose transposition is being done on 2 elements a and b. If a precedes b in both L_i and L_i^*, then this is not an inversion. Transposing a and b in L_i would lead to b preceding a in L_i and a preceding b in L_i^*, so the number of inversions increases by 1. If we transpose a and b in L_i^* then a will precede b in L_i and b will precede a in L_i^*, increasing the number of inversions by 1.

- Similarly, if b precedes a in both L_i and L_i^*, this is not an inversion. Transposing a and b in either L_i or L_i^* would increase the number of inversions by 1.
- If a precedes b in L_i and b precedes a in L_i^*, this is an inversion. Transposing them in L_i would make b precede a in both L_i and L_i^*, thus reducing the number of inversions by 1. Transposing them in L_i^* would make a precede b in both L_i and L_i^*, thus reducing the number of inversions by 1.
- Similarly, if b precedes a in L_i and a precedes b in L_i^*, this is an inversion. Transposing a and b in either L_i or L_i^* will decrease the number of inversions by 1.

Since $\Phi(i) = 2q_i$ where q_i is the number of inversions, when we perform a transposition The new q_i is either $q_i - 1$ or $q_i + 1$, so the new potential is either $2q_i - 2$ or $2q_i + 2$, that is $\Phi(i) - 2$ or $\Phi(i) + 2$. So a transposition either increases or decreases the potential by 2.

17-5(e)

The elements before x in L_{i-1} are either before x or after x in L_{i-1}^*. Set A comprises the elements before x in L_{i-1} and before x in L_{i-1}^*, while set B comprises the elements before x in L_{i-1} and after x in L_{i-1}^*. So, if we add these 2 sets, we get all the elements before x in L_{i-1}. Then rank of x in L_{i-1} is its position in L_{i-1}, i.e. the number of elements before x in $L_{i-1} + 1$. Hence, $\text{rank}_{L_{i-1}}(x) = |A| + |B| + 1$.

Similarly, the elements before x in L_{i-1}^* are either before x or after x in L_{i-1}. Set A comprises the elements before x in L_{i-1}^* and before x in L_{i-1}, while set C comprises the elements before x in L_{i-1}^* and after x in L_{i-1}. So, if we add these 2 sets, we get all the elements before x in L_{i-1}^*. Then rank of x in L_{i-1}^* is its position in L_{i-1}^*, i.e. the number of elements before x in $L_{i-1}^* + 1$. Hence, $\text{rank}_{L_{i-1}^*}(x) = |A| + |C| + 1$.

17-5(f)

Proof. With the ***move-to-front*** heuristic, access σ_i finds x and then moves it to the front of the list. So the relative ordering between x and the set of elements in front of it is changed, which causes a change in the number of inversions. If there is an element in front of x in L_{i-1} and after x in L_{i-1}^*, then after access σ_i the element is after x in both L_i and L_i^*. So this removes the inversion. The elements that belong to this category are in set B. So the number of inversions reduces by $|B|$ with access σ_i. Additionally, if there is an element before x in L_{i-1} and L_{i-1}^*, then after the transpositions with the *move-to-front* heuristic the element will be after x in L_i and remain before x in L_i^* (as the heuristic does not modify any ordering in L_{i-1}^*). So while there was no inversion with this element and x before, there is an inversion now, which increases the number of inversions by 1. The elements that belong to this category are in set A. So the number of inversions increases by $|A|$ with access σ_i.

Access σ_i in L_{i-1}^* leads to t_i^* transpositions. The maximum increase in the number of inversions this leads to is t_i^*.

Thus, the total number of inversions are increased by at most $|A| - |B| + t_i^*$ with access σ_i, and so the change in potential is at most $2(|A| - |B| + t_i^*)$, or

$$\Phi(L_i) - \Phi(L_{i-1}) \leq 2(|A| - |B| + t_i^*).$$

$\blacksquare$

17-5(g)

Proof. From part (b) of the question,

$$\begin{aligned} c_i &= 2 \cdot \text{rank}_{L_{i-1}}(x) - 1 \\ &= 2(|A| + |B| + 1) \\ &= 2|A| + 2|B| + 1 \end{aligned} \tag{1}$$

From part (c) of the question,

$$\begin{aligned} c_i^* &= \text{rank}_{L_{i-1^*}}(x) + t_i^* \\ &= |A| + |C| + t_i^* + 1 \end{aligned} \tag{2}$$

From part (f) of the question,

$$\begin{aligned} \Phi(L_i) - \Phi(L_{i-1}) &\leq 2(|A| - |B| + t_i^*) \\ &= 2|A| - 2|B| + 2t_i^* \end{aligned} \tag{3}$$

Now

$$\begin{aligned} & \hat{c_i} = c_i + \Phi(L_i) - \Phi(L_{i-1}) \\ \implies & \hat{c_i} \leq 2|A| + 2|B| + 1 + 2|A| - 2|B| + 2t_i^* && \text{From (1) and (3)} \\ & \quad = 4|A| + 2t_i^* + 1 \\ & \quad = 4|A| + 2t_i^* + 1 + 2t_i^* + 3 \\ & \quad = 4|A| + 4t_i^* + 4 \\ & \quad \leq 4|A| + 4|C| + 4t_i^* + 4 \\ & \quad = 4(|A| + |C| + t_i^* + 1) \\ & \quad = 4c_i^* && \text{From (2)} \end{aligned}$$

$\blacksquare$